

Date: 2019-09-04 Author: Ian Rash Index: #4

When dealing with a protocol where the source code and implementation are secret to us, we need to do certain kinds of testing to find out how to format data appropriately, as well as determine what can change behavior, what can be ignored, and what we need to pay attention to.

We want to poke at SessionID a bit, let's try to login repeatedly to get an idea of SessionIDs values. For this example, we login to the camera, then send a command, then print the SessionID.

```
0 = 0x00000001
1 = 0x00000002
2 = 0x00000003
3 = 0x00000004
4 = 0x00000005
5 = 0x00000006
6 = 0x00000007
7 = 0x00000008
8 = 0x00000009
```

[Done] exited with code=0 in 1.025 seconds

Since SessionID acts like a simple incrementer, we can easily write our own values. Getting an idea of how the camera might produce its SessionID fields may help us later to develop an anti-client.

Let's make an attempt to send another command after logging in, we first will try to replay a command's SessionID, then try to replay the command with a randomized SessionID. This will give us greater insight into what works, and what doesn't.

For this command I'll be using SystemInfo which I found while sniffing! This one is used by the client to get a list of some settings and version numbers. Playing around with the SessionID field shows the camera doesn't care about it. I tried 0x00000007, 0x11111117, and a couple other random numbers and they all seemed to work!

There were also a couple values in the 20 byte header that I couldn't decipher, so I wrote a fuzzer to see if there was anything interesting about them. The results I got were, strange, to say the least. I used this method to make a basic command, including the header.

```
# Class that contains the basic process for making a message to/from the camera
class XMMessage
    property type : UInt32
    property session_id : UInt32
    property unknown1 : UInt32
    property unknown2 : UInt16
```

```
property magic : UInt16
property size : UInt32
property message : String
```

#TODO: Allow for spoofing of size, for example changing size to say that its 32 bytes, when its 0 or something

```
def self.from_s(string)
  io = IO::Memory.new string
  m = XMMessage.new
  m.type = io.read_bytes(UInt32, IO::ByteFormat::LittleEndian)
  m.session_id = io.read_bytes(UInt32, IO::ByteFormat::LittleEndian)
  m.unknown1 = io.read_bytes(UInt32, IO::ByteFormat::LittleEndian)
  m.unknown2 = io.read_bytes(UInt16, IO::ByteFormat::LittleEndian)
  m.magic = io.read_bytes(UInt16, IO::ByteFormat::LittleEndian)
  m.size = io.read_bytes(UInt32, IO::ByteFormat::LittleEndian)
  m.message = string[20..string.size]
  m
end
```

```
def initialize(@type = 0x000001ff_u32, @session_id = 0_u32, @unknown1 = 0_u32, @unknown2 = 0_u16, @magic =
end
```

```
def magic1 : UInt8
  (magic & 0xFF).to_u8
end
```

```
def magic2 : UInt8
  (magic >> 8).to_u8
end
```

```
def make_header
  header_io = IO::Memory.new
  header_io.write_bytes(type, IO::ByteFormat::LittleEndian)
  header_io.write_bytes(session_id, IO::ByteFormat::LittleEndian)
  header_io.write_bytes(unknown1, IO::ByteFormat::LittleEndian)
  header_io.write_bytes(unknown2, IO::ByteFormat::LittleEndian)
  header_io.write_bytes(magic, IO::ByteFormat::LittleEndian)
  header_io.write_bytes(self.message.size, IO::ByteFormat::LittleEndian)
```

```
  header_io.to_s
end
```

```
def make : String
  (make_header + self.message)
end
end
```

Next, I made the main part of the fuzzer, which fills in every possible byte value, waits for a reply, then disconnects,

and loops. I keep track of which magic return what reply.

If you'd like to view the code, and understand more about how the fuzzing process needs to work, check out the [GitHub](#) page for the project.

Some notes on this program,

- There are two bytes we want to target, the first magic1, and then magic2.
- The fuzzer itself has to be extremely fault tolerant, because lots of bad things happen when using it. There is a lot of trial and error in designing one of these.

Was the login connection refused? (because camera went offline) Was the command ever replied to? * Was the command reply's length zero?

When I want to run it I just do something like this,

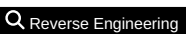
```
class Command::SystemInfo < Command
  def initialize(@session_id = 0)
    super(magic1: 0xfc_u8, magic2: 0x03_u8, json: JSON.build do |json|
      json.object do
        json.field "Name", "SystemInfo"
        json.field "SessionID", "0x#{@session_id.to_s(16).rjust(8, '0')}}"
      end
    end)
  end
end

File.open("logs/system_info.log", "w") do |file|
  Fuzzer.run(
    Command::SystemInfo.new(session_id: 0x1111117),
    magic2: (0x3..0x6),
    password: "password",
    output: file
  )
end
```

This will run for a while, eventually producing results, and while the results are accurate, it's too slow! It can take up to 24 hours to complete a single scan of the magic field 0x3 to 0x8, especially because camera turns off and doesn't turn back on for minutes, as well as many other system breaking bugs.

Luckily I had a couple ideas to make it faster.

TAGGED SKILLS:

 Fuzzing  Security Research  Reverse Engineering

